

Automatically finding atomic regions for fixing bugs in Concurrent programs

Saurabh Joshi¹ and Akash Lal²

¹ Department of CSE, IIT Kanpur, India

² Microsoft Research, India

Abstract. This paper presents a technique for automatically constructing a fix for buggy concurrent programs: given a concurrent program that does not satisfy user-provided assertions, we infer atomic blocks that fix the program. An atomic block protects a piece of code and ensures that it runs without interruption from other threads. Our technique uses a verification tool as a subroutine to find the smallest atomic regions that remove all bugs in a given program. Keeping the atomic regions small allows for maximum concurrency. We have implemented our approach in a tool called ATOMICINF. A user of ATOMICINF can choose between strong and weak atomicity semantics for the inferred fix. While the former is simpler to find, the latter provides more information about the bugs that got fixed.

We ran ATOMICINF on several benchmarks and came up with the smallest and the most precise atomic regions in all of them. We implemented an earlier technique to our setting and observed that ATOMICINF is 1.7 times faster on an average as compared to an earlier approach.

1 Introduction

An important part of the debugging process is to come up with a repair that fixes the bug under investigation. After a candidate repair is formulated, not only must one reason that the fix removes the bug, but also that it does not introduce new bugs in the program. Thus, evaluating a repair requires understanding of the program as a whole, not just the executions that reveal the current bug. Consequently, any automation in the process of formulating and evaluating a fix would be welcome.

At first, this debugging problem seems to be a good match for verification tools that are prepared to reason over many (or all) program behaviors. However, the process of formulating a fix can be difficult to automate. For instance, any program with an assertion `assert(!error)` can be “fixed” by inserting the statement `error=false` right before the assert. However, such repairs are clearly of no practical use.

In order to get across this challenge of meaningless repairs, we focus on a restricted problem of automated program repair. In particular, we focus on fixing concurrent programs by introducing extra synchronization that restricts the set of interleavings possible in the program. *Given a specification, we infer a*

fix that removes all of the bad interleavings of the program while minimizing the set of good interleavings that get removed by the fix. This problem definition has several advantages. First, a fix is not allowed to introduce new behaviors, i.e., any execution of the fixed program is also a valid execution of the original program. This rules out, for instance, the trivial repair mentioned previously. Second, we remove all bad interleavings, which implies that the resulting program will satisfy the specification. Third, by minimizing the set of good interleavings removed, we allow maximum concurrency in the program and avoid significantly degrading the performance and responsiveness of the program. Fourth, the specification is supplied by the user, allowing one to target the repair towards certain (user-defined) properties.

We restrict the space of interleavings by introducing extra synchronization in the program in the form of *atomic blocks*. Atomic blocks are a convenient way of expressing synchronization. Previous work shows that programs that use atomic blocks are easier to understand than ones that use locks [17]. An atomic block is used to enclose a piece of code that restricts how that code interacts with concurrently executing threads. The exact semantics depends on the type of atomic block used. A *strong* atomic block ensures that the enclosed code executes in complete isolation of the rest of the program. A *weak* atomic block ensures that the enclosed code executes in isolation of other weak atomic sections. We allow a user to pick which kind of atomic blocks to use for the fix. A fix using strong atomic blocks is easier to find, but a fix using weak atomic blocks usually reveals more information about the bug getting fixed. Furthermore, it is easy to realize weak atomic blocks using locks [4].

Our approach works as follows. We accept, as input, a program with assertions. We assume that the program specification is fully captured by the asserts. (Any safety property can be captured using assertions.) *Furthermore, we assume that all executions of the program in which threads do not interleave (i.e., the threads execute sequentially, one after the other) are correct.* This is an important assumption because otherwise, no set of atomic blocks could repair the program. Next, we use any off-the-shelf verification tool to iteratively reveal more and more buggy traces in the program until we converge on a fix. Because queries to the verification tool can be expensive, we minimize the number of buggy traces required.

When a user selects strong atomicity, we guarantee that the reported fix is the smallest in terms of the number of program points protected by the atomic block. However, the search for the smallest weak atomic blocks turns out to be too expensive. Thus, when a user selects weak atomicity, we employ a crucial optimization. We first find the smallest fix F under strong atomicity and then restrict the search for weak atomic blocks to those which are supersets of F . While this implies that the fix may not be the smallest, we still guarantee that it is a minimal extension of F . Furthermore, our experiments reveal that this optimization does not compromise the quality of the fix.

The key contributions of this paper are as follows:

- We give an efficient approach for finding a smallest fix F under strong atomicity as well as a minimal extension of F under weak atomicity.
- Our approach is completely driven by user-supplied properties, as opposed to previous work that relies on symptoms such as data races and atomicity violations to root-cause a bug [5, 18]. Hence, our technique does not get distracted by benign data races.
- Our experiments show that we are able to find the best fix on a variety of benchmarks.

The rest of the paper is organized as follows. Section 2 gives an overview of our technique. Section 2.1 describes the related work. Sections 3 to 5 describe our techniques in detail with algorithms and proofs. Section 6 mentions experimental results.

2 Overview

This section gives an overview of **ATOMICINF**. It accepts as input a concurrent program with assertions. We use the term *bug* to refer to an execution that ends in an assertion violation.

We consider two semantics for atomic blocks. A *strong atomic block* **satomic**(S) guards a region of code S and ensures that it runs in complete isolation with respect to other threads. Intuitively, this means that context switching is disabled while S is executing. A *weak atomic block* **watomic**(S) ensures that S runs in isolation with respect to other weak atomic blocks. One simple (semantics preserving) implementation of weak atomic blocks is to use a single global lock l and replace each block **watomic**(S) with (acquire(l); S ; release(l)). In this sense, it is much easier to realize weak atomic blocks in a language runtime and people have proposed efficient implementations for them [4].

Consider the **banking** program shown in Fig. 1. The procedure **transfer** transfers a given amount from one account to another account. The procedure **seize** sets the balance in a given account to zero. The thread **thread1** attempts to transfer 200 units from **acc1** to **acc2**; **thread2** tries to set the balance in **acc1** to zero; and **thread3** tries to transfer 100 units from **acc2** to **acc1**. The program is buggy: for example, immediately after **thread1** checks whether sufficient amount is available in **acc1** in **transfer**, **thread3** starts and runs to completion, then **thread2** also runs to completion, setting the balance in **acc1** to 0. Next, when **thread1** finishes, the amount in **acc1** is negative. The appropriate fixes for the program are the following: (1) enclose the body of **transfer** in a strong atomic block; or (2) enclose the bodies of both **transfer** and **seize** in weak atomic blocks. Note that the latter solution is more informative because it points out the conflicting concurrent accesses. **ATOMICINF** can find these fixes automatically.

Let **VERIFTOOL** refer to any verification tool that gives us the ability of controlling the places where context switches are allowed. Let P be the buggy concurrent program. We feed P to **VERIFTOOL** and get a trace t that witnesses a buggy execution of P . We examine the set of program locations where the trace

```

1  struct Account {
2      int amount;
3  } acc1, acc2, acc3;
4
5  void seize(Account *acc) {
6      acc->amount = 0;
7  }
8
9  void thread1() {
10     transfer(&acc1,&acc2,200);
11 }
12 void thread2() {
13     seize(&acc1);
14 }
15 void thread3() {
16     transfer(&acc2,&acc1,100);
17 }
18
19 int transfer(Account* src,Account* dst,int amount) {
20     if(src->amount>=amount) {
21         int temp = src->amount;
22         temp = temp - amount;
23         src->amount = temp;
24         temp = dst->amount;
25         temp = temp + amount;
26         dst->amount = temp;
27         return 1;
28     }
29     return 0;
30 }
31 void main() {
32     acc1.amount = acc2.amount = acc3.amount = 200;
33     t1 = async thread1();
34     t2 = async thread2();
35     t3 = async thread3();
36     join(t1); join(t2); join(t3);
37     assert(acc1.amount == 0 || acc1.amount == 100);
38 }

```

Fig. 1: The Banking example. The `async` keyword spawns a thread and returns a handle for the thread. The `join` command waits for the thread to terminate.

took context switches. Lets say this set is $\{l_1, l_2, l_3\}$. Then we add the constraint “ $\neg(l_1 \wedge l_2 \wedge l_3)$ ” to VERIFTOOL, i.e., we disallow it from considering traces that take context switches at all of these locations. This constraint rules out t from appearing again. Next, we feed P to VERIFTOOL under this constraint. We repeat this process until we get no more buggy traces. Suppose that L_i was the set of context switch locations of the i^{th} buggy trace. Then a solution, for strong atomicity, is to select at least one location from each L_i , and enclose all such locations in strong atomic blocks. This is sound because the fix disables all buggy traces of P . This technique is inspired from previous work [22]. As we describe later, this technique is very inefficient because the number of queries made to VERIFTOOL can be proportional to the size of the program, irrespective of the size of the solution. For example, suppose the fix requires protecting just one location l . Then there would still be lots of buggy traces that take a context switch at l but also take (possibly redundant) context switches at other locations. The above iterative process will enumerate all such traces.

We propose an algorithm where the number of queries to VERIFTOOL remains roughly proportional to the size of the solution. In order to accomplish this, we use the same iterative scheme as above, but after the i^{th} iteration, we take the i buggy traces and construct a “proposed” solution by selecting at least one context-switch location from each of the i traces. Next, we ask VERIFTOOL if this solution fixes the program. If so, we’re done. Otherwise, we get a buggy trace and we repeat the process. This way, redundant context switches get ruled out very quickly. For example, if the fix is one location l , then after getting two traces that take context switches $\{l, l_1\}$ and $\{l, l_2\}$, the proposed solution of $\{l\}$ will be the correct fix, without requiring enumeration of other buggy traces.

| | |
|---|---|
| <pre> 1 int transfer(Account src, Account dst, int amount) 2 { 3 satomic 4 { 5 if(src.amount >= amount) 6 { 7 int temp = src.amount; 8 temp = temp - amount; 9 src.amount = temp; 10 --> 11 temp = dst.amount; 12 temp = temp + amount; 13 dst.amount = temp; 14 return 1; 15 } 16 } 17 return 0; 18 }</pre> | <pre> 1 void thread2() 2 { 3 int temp=0; 4 satomic{ 5 temp = acc1.account; 6 temp += acc2.account; 7 assert(temp == 400); 8 } 9 } 10</pre> |
| 2(a) | 2(b) |

Fig. 2: 2(a) The strong atomicity fix found by ATOMICINF for the program in Fig. 1. 2(b) Asserting that the corpus of the bank must remain constant at all times.

For weak atomicity, we follow a similar iterative process, but mine the error traces for more information. In particular, we look for pairs of locations (l_1, l_2) such that a buggy trace takes a context switch at l_1 and passes through l_2 . We can rule out this error trace by placing both locations inside a weak atomic block. To reduce the search space, we first find a solution under strong atomicity and then use it as a starting point for finding a solution under weak atomicity. In many cases, these solutions are very similar, thus, reducing the number of iterations required for weak atomicity. For the **banking** example, once the strong atomicity fix is found, extending it to a weak atomicity fix only requires enclosing the **seize** procedure in a weak atomic block.

We now illustrate the property-guided nature of ATOMICINF. For the program of Fig. 1, the fix reported by ATOMICINF allowed a context switch at line 10, as shown in Fig. 2(a). On closer inspection, this is a valid solution; it says that operations of debiting **amount** from **src**, and crediting to **dst** need to be individually atomic, but it is fine for other operations to execute between them.

As a further test, we changed the implementation of the second thread to what is shown in Fig. 2(b). It checks that the corpus of money in the two accounts remains constant. Because this is done in a thread, the assertion can fire any time during the program’s execution. In this case, ATOMICINF proposes that the entire body of **transfer** needs to be inside a single strong atomic block; it is no longer safe to interleave operations between the debit and credit of **transfer**.

2.1 Related Work

Automatic repair of programs has been studied earlier, both for sequential programs [2, 6, 9, 13] as well as for concurrent programs [3, 8, 15, 21, 22]. Previous

work on sequential programs has focused on formulating program repair as a two-player game, where one of the players tries to make sure that the program doesn't fail. A winning strategy for this player is the repair. One limits the vocabulary of the player (to, for instance, memory-less players) in order to reduce the search space and come up with a reasonable fix. This work is orthogonal to ours because we do not try to repair sequential executions of a program.

For concurrent programs, a majority of the work uses dynamic analysis to repair bugs like atomicity violations [1, 8, 12]. For instance, RECON [12] uses test runs to locate bugs and then uses statistical analysis over these runs to infer a fix. Being dynamic in nature allows these techniques to scale, but the quality of the solution is dependant on the coverage of the test runs. On the other hand, our approach uses static analysis and is capable of providing soundness guarantees for the fix. Moreover, our notion of a bug is an assertion failure, not notions like data-races and atomicity violations. Thus, our approach does not get distracted by benign (and intended) data-races and atomicity violations.

Some techniques require user annotations to infer necessary synchronization. For example, the approach described in [20] infers synchronization once a user annotates sets of fields, indicating existence of a consistency property within members of each set.

A quantitative approach to synthesize synchronization has been proposed in [21]. This work tries to optimize the synthesis of synchronization with respect to a performance model. Though this work provides correctness as well as performance guarantees about the fix, it only works for finite-state programs, making its use very limited.

We now discuss two pieces of work that are most similar to ours. First is WYPIWYG (What-You-Prove-Is-What-You-Get) [3], which takes a correct sequential library and then synthesizes synchronization (in the form of locks) to make sure that the library functions correctly even in the presence of a concurrent client. Their idea is to take the proof-of-correctness under a sequential client and then construct synchronization to preserve the same proof even under a concurrent client. This approach contrasts with ours in the following ways: First, ATOMICINF relies on a bug-finding tool, not necessarily ones that can produce a proof of correctness. Second, ATOMICINF guarantees to find the smallest fix (under atomic sections) irrespective of the underneath verification tool, whereas the quality of the solution in WYPIWYG depends completely on the quality of the proof produced—the more modular the proof, the better the synchronization inferred. We ran ATOMICINF on the benchmarks used by WYPIWYG. Both approaches inferred the ideal synchronization. However, it is not possible to compare the running times because WYPIWYG used a manually-constructed proof of correctness for some benchmarks.

The work by Vechev et al. [22] is also very similar to ours. The goal of their work was to exhibit the power of abstraction-refinement for synthesizing synchronization using strong atomic blocks. We recast their approach to our setting in Section 4.1, and then show that our technique (Section 4.2) is more

efficient. Moreover, their work did not address inferring synchronization under weak atomicity.

3 Preliminaries

This section sets up the program syntax used in the rest of the paper and the problem definition. Because we want to control context switching in the program, we assume a co-operative model of concurrency where a program is only allowed to take a context switch at a special **yield** instruction. We write programs using C syntax, extended with the following constructs.

- yield** : The program can context switch only at this statement.
- assume**(e) : If the expression e evaluates to *false* then the program blocks, otherwise it continues to the next statement.
- axiom**(e) : This statement is similar to having **assume**(e) at all points in the program. We use **axiom** to insert global invariants into a program.
- satomic**($stmt$) : This specifies a strong atomic region. $stmt$ is executed atomically, i.e., no context switches are allowed while executing $stmt$.
- watomic**($stmt$) : This specifies weak atomic region. $stmt$ is executed in isolation with respect to all other **watomic** blocks. In other words, the execution of $stmt$ can not begin if some other thread is executing inside a **watomic** block.
- async** $m()$: This construct spawns a thread which executes method $m()$. It also returns a handle of the thread created.
- join**(tid) : This statement waits for the thread, represented by its handle tid , to terminate.

Using the co-operative model of concurrency is not restrictive. Given a multi-threaded program P , one can insert **yield** instructions before any instruction that accesses a shared memory location, and also as the first instruction of a thread. The resulting program, under co-operative semantics, is equivalent to P . For example, the left side of Fig. 3 shows how the **transfer** method of Fig. 1 is instrumented for co-operative semantics. (For simplicity, we assume that each line of code executes atomically.) Based on this model, we define the notion of a *minimum fix* as follows.

Definition 1. A *minimum fix* for a program P is one which encloses the least number of **yield** statements under strong or weak atomic blocks.

Sections 4 and 5 address problems of finding a fix under strong or weak atomicity semantics respectively.

Once we have a fix under the co-operative model, we map the fix back to one in the multi-threaded model. Let Y be the set of yield instructions that need to be protected by an atomic block, and let S be the set of original program locations where these instructions were inserted. Next, we say that two statements $stmt_1$ and $stmt_2$ are *connected* if there is a path from $stmt_1$ to $stmt_2$ or from $stmt_2$ to $stmt_1$ in the control flow graph of the program, such that this path does not

pass through any program point $p \notin S$. We compute such maximally connected components within the CFG and output it as the atomic blocks. These regions may not be lexically scoped. One way to make them lexically scoped is to consider the set of statements that falls between the dominator and the postdominator of the maximally connected component found earlier. It is a matter of choice whether to output a maximally connected component as a region or augment it to make it lexically scoped.

Limitations Although our algorithms guarantee to find the least number of program points to protect in a fix, the process of actually reporting atomic blocks may lose this guarantee; finding lexically-scoped blocks can force us to include other program points in the atomic blocks. However, this is not a major limitation. `ATOMICINF` also reports the collection of program points and it is usually easy to manually infer the desired fix from this collection of points.

Another limitation is that, in general, the fix inferred by `ATOMICINF` can only guarantee correctness with respect to safety properties. *It cannot handle liveness properties.* This limitation shows up when the input program itself has some synchronization. Then, imposing the fix inferred by `ATOMICINF` can lead to deadlocks. For instance, if the program uses flag-based synchronization via a loop: `while(!flag) { }`, (i.e., a thread waits for some other thread to set `flag` to true), then disabling context switches within the body of this loop can cause a deadlock. We circumvent this problem by never including **yield** instructions that are meant for synchronization in our fix. This is done partly automatic: **yield** statements before synchronization operations such as locking routines, and just after an **async** are excluded from the fix; and partly manual: a user annotates explicit **yield** points inside shared-memory based synchronization operations, which are also excluded from the fix. We leave a more detailed study for fixing liveness properties as future work.

4 Strong Atomicity Inference

Our first step is to gain control over context switching in the program. We do this by introducing a fresh Boolean constant for each **yield** instruction (except ones excluded because of synchronization—see Section 3), and then guard the **yields** using this constant as shown in Fig. 3. Let CSG be the set of Boolean constants introduced this way. Forcing a Boolean constant $cs_i \in CSG$ to be *false* will prevent the context switch from happening at corresponding **yield** point. For example, in Fig. 3 if we want `src->amount` to be decremented atomically, we add `axiom(cs3 == false)` to the program. We also use these Boolean constants to identify the location of a **yield** instruction.

Given a formula ϕ over CSG , let $\langle P, \phi \rangle$ be the program P extended with the statement `axiom(ϕ)`. If $S \subseteq CSG$, then let $disable(S) = \bigwedge_{cs_i \in S} \neg cs_i$. Our goal is to find the smallest set S such that $\langle P, disable(S) \rangle$ is a correct program.

For a trace t , let $CS(t) \subseteq CSG$ be the set of Boolean constants corresponding to the context switches taken in t . Note that $CS(t)$ cannot be empty when t is

| Before | After |
|--|--|
| <pre> int transfer(Account* src, Account* dst, int amount) { yield; if(src->amount >= amount) { yield; int temp = src->amount; temp = temp - amount; yield; src->amount = temp; yield; temp = dst->amount; temp = temp + amount; yield; dst->amount = temp; return 1; } return 0; } </pre> | <pre> int transfer(Account* src, Account* dst, int amount) { if(cs1) { yield; } if(src->amount >= amount) { if(cs2) { yield; } int temp = src->amount; temp = temp - amount; if(cs3) { yield; } src->amount = temp; if(cs4) { yield; } temp = dst->amount; temp = temp + amount; if(cs5) { yield; } dst->amount = temp; return 1; } return 0; } const bool cs1, cs2, cs3, cs4, cs5; </pre> |

Fig. 3: Transforming a program to guard yields

an error trace of a program without sequential bugs. As previously noted, we assume that the program does not have any sequential bugs. Let $\text{BTRACES}(P)$ be the (possibly infinite) set of all error traces of program P . Let $\text{CSTRACES}(P)$ be $\{\text{CS}(t) \mid t \in \text{BTRACES}(P)\}$. Thus, $\text{CSTRACES} \subseteq (\mathcal{P}(\text{CSG}) \setminus \{\emptyset\})$, where \mathcal{P} denotes the power set of a given set. Since CSG is finite, CSTRACES will be finite as well.

For a program P , a valid fix is one that rules out all traces in $\text{BTRACES}(P)$. To disallow a trace t , it is sufficient to disable any one of the context switches taken by t . Thus, a fix for P is to disable a set of context switches S such that S is a *hitting set* of $\text{CSTRACES}(P)$. And the smallest fix is a *minimum hitting set* (MHS) of $\text{CSTRACES}(P)$. Note that MHS of any collection of sets need not be unique.

Definition 2. Given a set U and a collection of sets $C \subseteq \mathcal{P}(U) \setminus \{\emptyset\}$, a set $H \subseteq U$ is a **hitting set** of C if $\forall S_i \in C, S_i \cap H \neq \emptyset$. Furthermore, H is called a **minimum hitting set** (MHS) if C does not have a smaller hitting set.

Finding an MHS is NP-complete, but for the problem instances that we generate, it is usually quite easy to find an MHS.

4.1 A First Approach [22]

Alg. 1 describes an initial approach for finding the smallest set S such that $\langle P, \text{disable}(S) \rangle$ is correct. This approach is inspired from the work of Vechev et al. [22]. Let VERIFTOOL be a verification tool. Given a program, $\text{VERIFTOOL}(P)$ returns $\text{BUG}(t)$ if P has a bug and the error trace is t , else it returns CORRECT .

Alg. 1 iteratively (lines 5-16) finds an error trace t and stores the set of context switches taken by it in C . Then ϕ is modified to make sure that the

Algorithm 1 Minimum Hitting Set Solution

```

1: input: Concurrent program  $P$  instrumented
   with Boolean guards for yields.
2: output: Set  $S$  of context switches, such that
    $\langle P, \text{disable}(S) \rangle$  is correct.
3:  $\phi := \text{true}$ 
4:  $C := \emptyset$ 
5: loop
6:    $\text{res} := \text{VERIFTOOL}(\langle P, \phi \rangle)$ 
7:   if  $\text{res} == \text{CORRECT}$  then
8:     break
9:   end if
10:   $\text{let } \text{BUG}(t) = \text{res}$ 
11:  if  $\text{CS}(t) == \emptyset$  then
12:    throw exception(“Program has a se-
      quential bug  $t$ ”)
13:  end if
14:   $\phi := \phi \wedge \left( \bigvee_{cs \in \text{CS}(t)} \neg cs \right)$ 
15:   $C := C \cup \{\text{CS}(t)\}$ 
16: end loop
17: return  $\text{MHS}(C)$ 

```

Algorithm 2 Optimized Minimum Hitting Set Solution

```

1: input: Concurrent program  $P$  instrumented
   with Boolean guards for yields.
2: output: Set  $S$  of context switches, such that
    $\langle P, \text{disable}(S) \rangle$  is correct.
3:  $\phi := \text{true}$ 
4:  $C := \emptyset$ 
5: loop
6:    $\text{res} := \text{VERIFTOOL}(\langle P, \phi \rangle)$ 
7:   if  $\text{res} == \text{CORRECT}$  then
8:     break
9:   end if
10:   $\text{let } \text{BUG}(t) = \text{res}$ 
11:  if  $\text{CS}(t) == \emptyset$  then
12:    throw exception(“Program has a se-
      quential bug  $t$ ”)
13:  end if
14:   $C := C \cup \{\text{CS}(t)\}$ 
15:   $\phi := \text{disable}(\text{MHS}(C))$ 
16: end loop
17: return  $\text{MHS}(C)$ 

```

same trace t does not manifest again (line 14) by disallowing at least one of the context switches taken by t . This is repeated until no more bugs are found. Alg. 1 returns a solution of the smallest size by computing the MHS of C .

```

x = 10;
if(cs1) yield();
assert(x == 10);

if(cs2) yield();
x = 5;
if(cs3) yield();
tmp = 1;
if(cs4) yield();
tmp = 1;
...
if(csN) yield();
tmp = 1;

```

Fig. 4: A code snippet.

As mentioned in Section 2, this algorithm is not very efficient. Consider the code snippet shown in Fig. 4. Suppose there are two threads executing this code. The first thread executes the code on the left and the second thread executes the code on the right. The program fails whenever the statement $x = 5$ gets interleaved between statements $x = 10$ and the assertion. Assignments to `tmp` are redundant but they introduce extra yield points.

When we run Alg. 1 on this code, we can get error traces that first execute $x=10$, then context switch at `cs1`, then execute $x=5$ and some part of the second thread, context switch at `csi` (for $3 \leq i \leq N$), and then fail the assertion. There can be $N - 1$ such traces. Thus, Alg. 1 will potentially make $N - 1$ calls to VERIFTOOL. While the fix is to disable just `cs1`, the number of verification calls made by this approach is proportional to the size of the program.

4.2 Our Approach

Alg. 2 improves the previous algorithm by being more efficient when the size of the solution is small. The main difference is on line 15. It computes a proposed solution by looking at all previous traces. To see why this is an improvement, let us again consider the program in Fig. 4. Suppose the first trace takes context switches $S_1 = \{cs_1, cs_5\}$. Then $C = \{S_1\}$ and it has two possible choices of MHSs. Suppose (unluckily) we pick $MHS(C)$ as $\{cs_5\}$. Then ϕ disables cs_5 . The VERIFTOOL call will return another error trace passing through, say, $S_2 = \{cs_1, cs_8\}$ (note that all error traces have to take cs_1). Then $C = \{S_1, S_2\}$ and has exactly one MHS, which is $\{cs_1\}$. Thus, we converge to the desired solution in just two queries, independent of N . Furthermore, the constraints ϕ added to the program P are much simpler than the ones added by Alg. 1, making the job of the verifier easier.

Theorem 1. *Given a program P with no sequential bugs, Alg. 1 and Alg. 2 compute a minimum hitting set of $CSTRACES(P)$.*

Proof. Let m be the MHS of $CSTRACES$. Each of the algorithms returns an MHS over some subset of $CSTRACES$. Let C_i be the subset used by Alg. i and let m_i be its MHS. Both m_1 and m_2 are valid fixes because VERIFTOOL eventually returns CORRECT. Thus, both are hitting sets of $CSTRACES$. Because m is an MHS of $CSTRACES$, it must be a hitting set of C_1 . This implies $|m_1| \leq |m|$. Thus, m_1 is an MHS of $CSTRACES$. Same argument applies for m_2 .

Performance comparison between Alg. 1 and Alg. 2 : A direct theoretical comparison between the running times of Alg. 1 and Alg. 2 is difficult because of inherent non-determinism in these algorithms. In particular, the verification tool may return any arbitrary buggy trace in the program fed to it, making it possible for any of Alg. 1 and Alg. 2 to get "lucky" and converge to a fix faster. However, we can show that if both algorithms witness the same set of traces, then Alg. 2 is never worse than Alg. 1.

Let ϕ^{alg1} and ϕ^{alg2} denote the constraints generated by Alg. 1 and Alg. 2 respectively, on lines 14 and 15. Further, suppose that the first n iterations of the algorithms witness the same traces t_1, \dots, t_n . Then it must be that ϕ^{alg2} is stronger than ϕ^{alg1} : For every trace t_i , ϕ^{alg1} has a clause $(\bigvee_{cs \in CS(t_i)} \neg cs)$. On the other hand, ϕ^{alg2} has a clause with a single literal $\neg cs'$, where, cs' is the context switch taken by t_i and is a part of an MHS computed by it. Then $\phi^{alg2} \rightarrow \phi^{alg1}$ follows from $a \rightarrow a \vee b$ (for each clause corresponding to a trace) as well as $a \rightarrow b \wedge c \rightarrow d \Rightarrow a \wedge b \rightarrow c \wedge d$ (conjunction of clauses from all the traces). Consequently, if Alg. 1 terminates in the $n + 1^{st}$ iteration, then so will Alg. 2. Our experiments (Section 6) show the superiority of Alg. 2 in practice.

5 Weak atomicity inference

Computing a fix using weak atomicity is harder because it doesn't directly allow us to disable context switches. We set up some terminology first.

Definition 3. Given a trace t and a context switch cs taken by t , let T be the thread that was executing when cs was taken. Then the **lifespan** of cs in t is defined as the set of all instructions (or program points) on t after cs but before T got control back. In other words, the lifespan of a context switch is the contiguous sub-trace between the two instructions of the same thread that surround the context switch.

For example, suppose $t = [a_1; a_2; a_3; b_1; b_2; c_1; c_2; b_3; a_4]$, where a_i denote instructions of thread 1, b_i denote instructions of thread 2, and c_i denote instructions of thread 3. Then the lifespan of the context switch at a_3 is $\{b_1, b_2, c_1, c_2, b_3\}$.

The way to rule out an error trace t using weak atomicity is to pick two yield instruction y_1 and y_2 on the trace such that: (1) y_1 appears before y_2 ; (2) t context switches at y_1 and (3) the lifespan of the context switch at y_1 includes y_2 . In this case, we say that y_2 *conflicts* with y_1 . Then including both y_1 and y_2 in a weak atomic block will render t infeasible. Moreover, this is the only way to disable a trace using weak atomic blocks (without inserting or deleting extra code). In contrast, for strong atomicity, we only had to look at y_1 . Thus, weak atomicity forces us to identify the conflict between threads.

As before, we introduce a Boolean constant for every yield instruction in the program. Furthermore, we introduce a global Boolean variable `lock` that is initialized to *false*. If cs is the

Boolean constant associated with a yield, then we transform it as follows:

$$\text{yield;} \Rightarrow \begin{cases} \text{if } (\neg cs) \{ \text{assume } lock == \text{false}; lock = \text{true}; \} \\ \text{yield;} \\ \text{if } (\neg cs) \{ lock = \text{false}; \} \end{cases}$$

This way, setting a Boolean constant to *false* is as if the corresponding `yield` is included in a weak atomic block.

For a trace t , let $WCS(t)$ be the set of (cs_1, cs_2) pairs such that cs_i corresponds to a yield instruction y_i , and y_2 conflicts with y_1 . Let $WCSTRACES(P)$ be $\{WCS(t) \mid t \in BTRACES(P)\}$. The smallest solution is given by the MHS of $WCSTRACES(P)$. If this set is W , then the following set of yields need to be protected by a weak atomic block: $\{y \mid (y_1, y) \in W \text{ or } (y, y_2) \in W\}$.

We can now set up our algorithm in a similar fashion to Alg. 2. However, we now have to gather pairs of instructions, which can lead to a large number

Algorithm 3 Conflict-Based Weak Atomicity Inference

```

1: input: Concurrent Program  $P$  instrumented
   with Boolean guards and lock, and strong
   atomicity solution  $S$ 
2: output: Weak atomicity solution.
3:  $\phi := \text{disable}(S)$ 
4:  $C := \{\}$ 
5: loop
6:    $res := \text{VERIFTOOL}(\langle P, \phi \rangle)$ 
7:   if  $res == \text{CORRECT}$  then
8:     break
9:   end if
10:  let  $\text{BUG}(t) = res$ 
11:  if  $WCS(t) == \emptyset$  then
12:    throw exception("Program has a sequential bug  $t$ ")
13:  end if
14:   $C := C \cup \{WCS(t, S)\}$ 
15:   $\phi := \text{disable}(MHS(C) \cup S)$ 
16: end loop
17: return  $S \cup MHS(C)$ 

```

of iterations. So we make use of a crucial optimization: First, we compute the strong atomicity solution $S \subseteq CSG$ for the program. Next, we only attempt to find the smallest extension of this solution that will fix the program using weak atomicity blocks. This is done as follows: For a trace t , instead of using $WCS(t)$, we use $WCS(t, S) \stackrel{\text{def}}{=} \{cs_2 \mid \exists cs_1 \in S : (cs_1, cs_2) \in WCS(t)\}$. Note that for an error trace t , if $WCS(t)$ is not empty then neither is $WCS(t, S)$ because we know that some context switch taken by t belongs in S . Thus, we only look for conflicts with context switches in the strong atomicity fix. Because $WCS(t, S)$ is a subset of CSG , we are again back to iterating over CSG rather than $CSG \times CSG$. Alg. 3 formalizes this description.

The penalty of using this optimization is that we do not guarantee the smallest fix, however, we do guarantee the smallest extension to the strong atomicity fix, and in our experiments we always obtained the smallest fix possible.

6 Implementation and Experiments

We have implemented Algs. 2 and 3 in a tool called **ATOMICINF**. We use **POIROT** [10,16] as the underlying verification tool. **POIROT** is really a bug-finding tool; it searches over all behaviors up to a bounded number of context switches, thus, it cannot prove the absence of bugs. In this case, the fix returned by **ATOMICINF** is correct only up to the capabilities of **POIROT**. In our experiments, we manually verified that the the computed fixes were sound. In principle, we could have used a true verification tool like **THREADER** [7] inside **ATOMICINF** to obtain sound fixes.

Results We evaluate the effect of changing various parameters on the performance of Algs. 1, 2, and 3. Consider the parameterized program shown in Fig. 5. It has two threads: the first executes the code on the left and the second thread executes the code on the right. The program has three parameters p_1, p_2 , and p_3 that control the program size. Note that the strong atomicity fix is to enclose the entire body of the first thread in an atomic block. Thus, the size of the strong atomicity fix is $p_1 + 1$ (the number of yields inside this block of code). The size of weak atomicity fix is $p_1 + p_2$ because all of the assignments to x in the second thread must be put inside a weak atomic block as well. The parameter p_3 controls the number of irrelevant assignments to shared variables. Results are shown in Fig. 6. Here, **#CS** is the number of yield instructions inserted in the program, **#Q** indicates the number of queries made to **POIROT** and the last column indicates running time in seconds. Compare Fig. 6(a) with Fig. 6(c). As expected, Alg. 1 requires more calls to **POIROT** as the size of the program increases. However, the number of calls made by Alg. 2 remains constant irrespective of the program size. Figs. 6(b) and 6(d) show that the number of queries required by Algs. 2 and 3 increases almost linearly as the size of the solution increases. Here, **(W)** in columns indicates the numbers for weak atomicity fix.

Next, we ran **ATOMICINF** on various benchmarks gathered from previous work. The results are shown in Tab. 1. In the table, **LOC** is lines of code, **#CS**

```

x = 10;
[y = 1;]p1
assert(x == 10);
||
[x = 1;]p2
[y = 1;]p3

```

Fig. 5: A parameterized program with two shared variables: x and y . Here, $[\text{st}]^n$ denotes the statement st repeated n times.

| p_3 | #CS | #Q | t(sec) |
|-------|-----|----|--------|
| 0 | 5 | 2 | 2.3 |
| 10 | 15 | 2 | 2.6 |
| 20 | 25 | 2 | 3.0 |
| 30 | 35 | 2 | 3.2 |
| 40 | 45 | 2 | 3.6 |

6(a) Changing Program Size(p_3) with Alg. 2 with $p_1 = 0, p_2 = 1$

| p_3 | #CS | #Q | t(sec) |
|-------|-----|----|--------|
| 0 | 5 | 3 | 2.7 |
| 10 | 15 | 13 | 7.7 |
| 20 | 25 | 23 | 16.2 |
| 30 | 35 | 33 | 27.0 |
| 40 | 45 | 43 | 41.4 |

6(c) Changing Program Size(p_3) with Alg. 1 with $p_1 = 0, p_2 = 1$

| p_2 | #CS | #Q | t(sec) | #Q(W) | t(W)(sec) |
|-------|-----|----|--------|-------|-----------|
| 4 | 8 | 3 | 2.9 | 8 | 5.1 |
| 8 | 12 | 3 | 3.0 | 12 | 8.5 |
| 12 | 16 | 3 | 3.3 | 16 | 11.7 |
| 16 | 20 | 3 | 3.5 | 20 | 15.8 |
| 20 | 24 | 3 | 3.7 | 24 | 20.8 |

6(b) Changing p_2 , keeping $p_1 = 0, p_3 = 0$

| p_1 | #CS | #Q | t(sec) | #Q(W) | t(W)(sec) |
|-------|-----|----|--------|-------|-----------|
| 0 | 5 | 2 | 2.3 | 4 | 2.9 |
| 2 | 7 | 7 | 4.3 | 9 | 5.1 |
| 4 | 9 | 11 | 5.9 | 13 | 6.9 |
| 6 | 11 | 15 | 8.0 | 17 | 8.8 |
| 8 | 13 | 19 | 10.1 | 21 | 11.2 |

6(d) Changing p_1 , keeping $p_2 = 1, p_3 = 0$

Fig. 6: Effects of changing various parameters of the program in Fig. 5

is the number yield instructions inserted in the program, Sol Size is the number of program points as part of the computed fix, #Queries is the number of times POIROT was called and the last column is the running time in seconds. The sub-columns $S1$ and $S2$ indicates results for strong atomicity by Alg. 1 and Alg. 2 respectively. W indicates weak atomicity results obtained by running Alg. 2 followed by Alg. 3. Numbers in bold indicates the better results amongst Alg. 1 and Alg. 2. Against each benchmark, we refer to the paper from which it was obtained. Here, *banking_inpaper* is the running example used in Fig. 1. Both the algorithms converged to the same solution for strong atomicity for all the examples. On the average Alg. 1 takes 20% more queries and 74% more time as compared to Alg. 2. If we discount for the outlier benchmark "BankAccount", Alg. 1 requires twice the number of queries on the average. As mentioned in Section 4.2 non-determinism plays a role as the two algorithms witness different set of traces and takes different amount of time. It is important to note that the most expensive operation in terms of time is a call to VERIFTOOL. We have observed that most of the time is spent inside the subroutine VERIFTOOL. Compared to this, the time consumed in computing MHS is negligible. For all of the examples, we manually inspected as well as cross verified with the papers from which the benchmarks were taken. We found the quality of the solution proposed by ATOMICINF to be the smallest and precise. On the other hand, for programs

| Example | LOC | #CS | Sol Size | | # Queries | | | Time(sec) | | |
|------------------------------------|-----|-----|----------|----|------------|-----------|-----|--------------|---------------|--------|
| | | | S | W | S1 | S2 | W | S1 | S2 | W |
| banking_inpaper(fig.1) | 62 | 22 | 2 | 3 | 22 | 6 | 9 | 35.9 | 12.3 | 20.8 |
| banking_inpaper_corpus (fig. 2(b)) | 58 | 23 | 3 | 4 | 12 | 7 | 9 | 16.6 | 11.4 | 13.5 |
| apache1 [19] | 64 | 10 | 2 | 2 | 4 | 3 | 4 | 4.5 | 4.0 | 4.3 |
| mozilla1 [11] | 64 | 7 | 2 | 2 | 4 | 3 | 4 | 3.7 | 3.3 | 3.6 |
| mysql [15] | 70 | 12 | 4 | 6 | 14 | 13 | 16 | 9.4 | 9.3 | 10.8 |
| banking [23] | 231 | 52 | 4 | 4 | 36 | 23 | 24 | 298.5 | 187.3 | 226.6 |
| defrag [22] | 142 | 37 | 2 | 2 | 56 | 5 | 6 | 475.5 | 187.6 | 2384.1 |
| doubleLockQueue [14] | 144 | 29 | 4 | 4 | 8 | 7 | 8 | 321.4 | 503.7 | 559.8 |
| jsClearMessagePane [12] | 311 | 68 | 2 | 4 | 51 | 7 | 12 | 854.3 | 175.0 | 489.2 |
| jsInterpBufferBool [12] | 215 | 36 | 1 | 2 | 23 | 3 | 15 | 85.6 | 16.3 | 65.9 |
| BankAccount [12] | 149 | 32 | 12 | 14 | 465 | 491 | 494 | 4306.0 | 2249.1 | 2546.5 |
| CircularList [12] | 139 | 29 | 8 | 9 | 9 | 9 | 11 | 232.8 | 229.5 | 669.0 |
| StringBuffer [12] | 126 | 25 | 11 | 12 | 45 | 48 | 50 | 202.4 | 205.0 | 524.8 |
| logProcessNSweep [12] | 149 | 26 | 3 | 4 | 26 | 23 | 26 | 371.3 | 332.8 | 474.5 |
| compute [3] | 51 | 7 | 2 | 2 | 6 | 5 | 6 | 9.7 | 8.8 | 14.9 |
| average [3] | 69 | 14 | 9 | 9 | 24 | 24 | 25 | 16.3 | 16.4 | 17.4 |
| increment [3] | 27 | 5 | 1 | 1 | 2 | 2 | 3 | 3.2 | 3.1 | 3.4 |
| nonDetRet [3] | 49 | 26 | 3 | 3 | 8 | 6 | 7 | 13.2 | 11.7 | 12.0 |

Table 1: Results of running ATOMICINF on a number of program snippets with published concurrency bugs.

logProcessNSweep and *CircularList*, the original approach [12] proposes a fix that includes 1 and 3 extra program points, respectively, which are not relevant to the bug being fixed.

References

1. Jacob Burnim, George C. Necula, and Koushik Sen. Specifying and checking semantic atomicity for multithreaded programs. In *ASPLOS*, pages 79–90, 2011.
2. Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *ICSE*, pages 121–130, New York, NY, USA, 2011.
3. Jyotirmoy Deshmukh, G. Ramalingam, Venkatesh-Prasad Ranganath, and Kapil Vaswani. Logical concurrency control from sequential proofs. In Andrew Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 226–245. 2010.
4. Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL*, pages 291–296, New York, NY, USA, 2007.
5. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, New York, NY, USA, 2004.
6. Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 358–371. 2006.
7. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, pages 412–417, 2011.
8. Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400, New York, NY, USA, 2011.
9. Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. 2005.

10. Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *CAV*, 2012. To appear.
11. Brandon Lucia, Luis Ceze, and Karin Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA*, pages 222–233, 2010.
12. Brandon Lucia, Benjamin P. Wood, and Luis Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI*, pages 378–388, 2011.
13. M.Z. Malik, J.H. Siddiqi, and S. Khurshid. Constraint-based program debugging using data structure repair. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 190–199, march 2011.
14. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, New York, NY, USA, 1996. ACM.
15. A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO*, pages 287–297, dec. 2010.
16. Poirot: The Concurrency Sleuth. <http://research.microsoft.com/en-us/projects/poirot/>.
17. Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPOPP*, pages 47–56, 2010.
18. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.
19. Aditya Thakur, Rathijit Sen, Ben Liblit, and Shan Lu. Cooperative crug isolation. In *WODA09*, pages 35–41, 2009.
20. Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, 2006.
21. Pavol Černý, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *CAV*, pages 243–259, Berlin, Heidelberg, 2011.
22. Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, New York, NY, USA, 2010.
23. Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, pages 256–272, Berlin, Heidelberg, 2009.

This figure "htraces.jpeg" is available in "jpeg" format from:

<http://arxiv.org/ps/1403.1749v1>

This figure "pst.jpeg" is available in "jpeg" format from:

<http://arxiv.org/ps/1403.1749v1>

This figure "pstbanking.jpeg" is available in "jpeg" format from:

<http://arxiv.org/ps/1403.1749v1>

This figure "traces1.jpeg" is available in "jpeg" format from:

<http://arxiv.org/ps/1403.1749v1>